Software Development with C++ Templates (541)

Lab Submission 1

It is suggested that exercises are solved in groups of two. However, it is also possible to solve them alone or in groups of three. Generally, observe that the bigger the size of the groups the more careful the exercises have to be implemented. Groups of three students have to solve optional exercises as mandatory ones.

Each exercise should be implemented together with a small program testing the individual functions implemented for the exercise. Those test cases should also check error handling, such as division by zero, etc.

Several exercises build on top of each other. Before continuing to implement the successor, please create a copy of the previous example, so that each step can be shown during the exercise submission.

All exercises should be commented using comments within the source code. Comments are only necessary when the code is not self explanatory. It is not necessary to document a single variable assignment in the constructor or the beginning of a function with "initialization." On the other hand, if the structure of a function is not clear and cannot be split up into self speaking sub-functions, a short comment should be inserted indicating the individual phases.

As coding style, following the Google C++ Coding style is suggested. In any case, the coding style should be consistent.

Exercise 1.1: A `fraction` class

Implement a `fraction` data type that provides the following operations. These operations should be implemented by overloading the respective C++ arithmetic operators ('+', '-', '*', and '/') as well as input and output routines. Ensure that all fractions are presented in normalized form. Implement a small test driver that shows the operation of your `fraction` class.

Exercise 2.1: Declaration vs. Definition

Which of the following lines belong into a header file? Explain why.

```
char ch;
string s;
extern int error_number;
static double sq(double);
int count=1;
const double pi=3.5; // ;)
struct fraction { int c; int d; };
char *name="It's me";
char *prog[]={"echo","hello","world!",NULL};
extern "C" void c_swap(int *a, int *b);
double sqrt(double);
void swap(int &a, int &b) {  int c=a; a=b; b=c;  }
```

```
namespace NS { int a; }
struct user;
```

Exercise 2.2: The `fraction` class with Separate Compilation

Extend the fraction class from exercise 1.1 to make use of multiple implementation and header files. Add a `Makefile` that compiles and links the indivudal files. The implementation of the `fraction` class should go into `fraction.cc`/`fraction.h` files, helper functions into `util.cc`/`util.h` files and the driver for testing into the file `fraction-test.cc`.

Exercise 2.3: RPN

Implement a reverse polish notation calculator. An RPN calculator is implemented by using a stack. When the user enters the command "n" followed by a number, the number is pushed onto the stack. The stack may be "indefinitely" deep. If the user enters a binary mathematical operator, two numbers are taken from the stack, the operation applied and the result pushed back onto the stack. Use the `std::vector` class as the implementation of your stack. When the user enters the command "d", a number is popped from the stack and discarded. When the user enters the command "q" the RPN calculator is terminated. Commands may be entered in any order.

The following shows how the application should run (user input in bold):

```
Command: n 2
1: 2
Command: n 4
1: 2
2: 4
Command: n 3
1: 2
2: 4
3: 3
Command: *
1: 2
2: 12
Command: -
1: -10
Command: d
Command: q
```

Exercise 2.4: Spell Checker

Implement a simple spell checker. The spell checker takes two files as command line arguments; the first being the filename of a dictionary containing a list of correctly spelled words and the second being the filename of the file whose content is to be checked.

Upon start, your program stores the words contained in the dictionary file in a `map<string>`. Next, it reads the word in the file to be spell-checked and checks for

each word whether it is spelled correctly (ie contained in the dictionary file) and if not, displays it.

Given the dictionary file dict.txt:

```
correctly spelled words are stored in this dictionary file
```

And the following file text.txt:

```
A comprehensive dictionary is important.
```

The invocation with spell dict.txt text.xt would produce the following output:

```
a
comprehensive
is
important
```

Exercise 3.1: Inline

Implement a program that shows a function that cannot be inlined. Are there other types of functions that cannot be inlined?

Exercise 3.2: Persistent Vector

Implement a persistent vector data type (`pvector`) similar to the one presented in the lecture using templates. Use the persistent vector in combination with different data types such as integers, floating point numbers, different types of strings, etc. What do you observe for the different data types? Explain the behavior. Implement a set of small demo programs.

Exercise 3.3: RPN Calculator with Persistent Vector

Extend your RPN calculator to make use of the persistent vector data type. That is, when the user terminates the RPN calculators with numbers left on the stack, they should reappear when the calculator is started again.

Exercise 3.4: RPN Calculator with Templates

Extend the RPN calculator from exercise 3.3 to support templates. It should be possible to instantiate the RPN calculators with int as data type and then the calculator makes use of integers for all calculations. Likewise it should be possible to instantiate the calculator with double or fraction and then the calculator should use these respective data types for all calculations. It is sufficient if the data type to be used is set in the source code before the code is compiled. Ensure that the data type only has to be changed in a single line within your source code.

Exercise 4.1: Persistent Vector with Traits

Adapt the implementation of your persistent vector to make use of traits like presented in the lecture. Adapt the demo programs from the previous exercises.

Implement a persistent set (`pset`) similar to the persistent vector data type which uses the same traits for persisting its elements.

Exercise 4.2: Interactive Dictionary

Extend your dictionary program to be interactive. Instead of printing out all unknown words, your program should allow the user to correct them or if the word was spelled correctly to insert the word into the dictionary. Make use of the `pset` implementation from the previous exercise.

Exercise 4.3: RPN Calculator with Standard Library Algorithms

Extend the last version of the RPN calculator to include standard library algorithms. As the minimum, include the `std::for_each` algorithm such when the user types the command 'm' that it puts a copy of the smallest number currently on the stack onto the top of the stack.

Exercise 4.4: Combineops

Explain the purpose of the following function object. Where could it be useful? Implement a program that demonstrates its use.

```
template <class BinOp, class Op1, class Op2>
class combineops_t :
public unary_function<typename Op1::argument_type,
                      typename BinOp::result_type> {
protected:
  BinOp o; Op1 o1; Op2 o2;
public:
  combineops_t(BinOp binop, Op1 op1, Op2 op2)
    : o(binop), o1(op1), o2(op2) {}
  typename BinOp::result_type
  operator()(const typename Op1::argument_type &x) {
    return o(o1(x),o2(x));
  }
};
```

Exercise 4.5: Template-Based Connect 4

Implement a template based version of the Connect 4 game where players. Connect 4 builds on a playing field composed out of 7 columns each having 6 rows. When a player puts a stone into a column, gravity pulls the stone towards the lowest unoccupied column. The player who first has 4 stones in a row (horizontally, vertically, diagonally) wins.

After each turn display the game field using simple ASCII graphics. Implement the game in such a way that players can be exchanged easily using templates. The precise interfaces to follow will be published at the lecture's homepage. It is important that you follow these interfaces religiously.

Exercise 4.6: Template-Based Connect 4 with Computer Player

Implement a computer player. The computer player of this version does not have to be intelligent. At a minimum, however, the computer player should be able to identify whether he can win the game by placing a stone. Let your computer player compete against computer players from your colleagues.

Exercise 5.1: Rectangles and Squares

Two computer scientists are implementing a vector based graphics program and are discussing whether a rectangle class should be inherited from a square class or whether the square class should be inherited from the rectangle class. The classes should have methods typically associated with such a kind of classes such as setting and getting the length of the sides, etc. Methods may be virtual or non-virtual. How, would you suggest that such classes are to be implemented? Implement different variations of such classes that demonstrate why your solution is better.